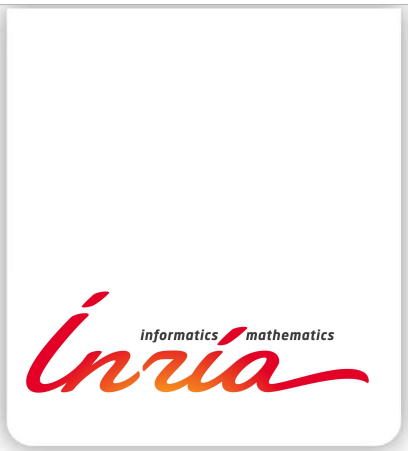
Handling Constraints in Cardinality-Based Feature Models: The Cloud Environment Case Study

Clément Quinton, Daniel Romero, Laurence Duchien

RESEARCH REPORT
N° 8478
February 2014
Team Spirals

ISSN 0249-6399 ISRN INRIA/RR--8478--FR+ENG



Handling Constraints in Cardinality-Based Feature Models: The Cloud Environment Case Study

Clément Quinton, Daniel Romero, Laurence Duchien

Team Spirals

Research Report n° 8478 — February 2014 — 22 pages

Abstract: Feature modeling is a well-known approach to describe variability in Software Product Lines. Cardinality-based Feature Models (FMs) is a type of FMs where features can be instantiated several times in the configuration, contrarily to boolean FMs where a feature is present or not. While boolean FMs configuration is easily handled by current approaches, there is still a lack of support regarding cardinality-based FMs. In particular, expressing constraints over the set of feature instances is not supported in current approaches, where cardinality involved in such constraints can not be specified. To face this limitation, we define in this paper cardinality-based expressions and provide the related formal syntax and semantics as well as the way to automate the underlying configuration. We study the need for such a support using cloud computing environment configurations as a motivating example. To evaluate the soundness of the proposed approach, we analyze a corpus of 10 cloud environments. Our empirical evaluation shows that constraints relying on our cardinality-based expressions are common and that our approach is effective and can provide an useful support to developers for modeling and reasoning about FMs with cardinalities.

Key-words: Variability Modeling, Constraint Expression, Feature Model, Software Product Line, Cloud Computing

RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Gestion des contraintes dans les modèles de caractéristiques avec cardinalités: étude de cas pour l'informatique dans les nuages

Résumé : La modélisation à l'aide de caractéristiques est une approche très utilisée dans les lignes de produits logiciels. Les Modèles de Caractéristiques (MCs) étendus avec des cardinalités sont un des MCs dans lesquels une caractéristique peut être instanciée plusieurs fois lors de la configuration, contrairement au MCs booléens dans lesquels une caractéristique est présente ou non. Alors que la configuration de MCs booléens est aujourd'hui maitrisée par différentes approches, il reste cependant un manque en terme de support pour les MCs étendus avec des cardinalités. Notamment, pouvoir exprimé des contraintes sur le nombre d'instances requises n'est pas permis dans les approches existantes, puisque les contraintes ne peuvent être exprimées que sur des caractéristiques booléennes. Pour contrer cette limite, nous fournissons dans cet article une nouvelle notation pour exprimer ces contraintes, une définition formelle de leur syntaxe et de leur sémantique ainsi qu'un moyen d'automatiser la vérification des configurations associées. Pour illustrer notre approche, nous étudions le besoin pour un tel support dans le cadre de la configuration d'environnements d'informatique dans les nuages. Nous évaluons notre approche sur un ensemble de 10 environnements. Notre étude empirique montre que les besoins pour exprimer ce type de contraintes sont communs dans ces environnements et que notre approche est efficace pour les gérer.

Mots-clés : Variabilité, Expression de Contraintes, Modèle de Caractéristique, Ligne de Produits Logiciels, Informatique dans les Nuages

Contents

1 Introduction

Software Product Line (SPL) engineering consists in the production of related software variants for a domain. SPL development begins with the description, management and implementation of the commonalities and variabilities existing among the members of the same family of software products [10, 26]. A well-known approach to variability modeling is by means of *Feature Model* (FM) [7] introduced as part of *Feature-Oriented Domain Analysis* (FODA) [22] back in 1990. In *Feature-Oriented Software Development* [5], a FM is an abstraction to define a software system where software artifacts are reified in the FM as features. FMs thus describe the way software artifacts are configured and reused to yield software products that satisfy a set of defined constraints. In these FMs, known as boolean FMs, a feature is either present or absent in the final product according to the configuration and the involved constraints.

However, despite their widespread use, conventional FMs are not sufficient in practice, where a final product often requires several instances of the same feature to run properly. For example, the configuration of cloud computing environments [6, 9] implies the definition of the number of running virtual machines, application servers or miscellaneous services. In our recent work and experiences in the configuration of such cloud environments using the *SoftwAre product Line for clOud cOmputiNg* (SALOON) framework [27, 29], we thus required a support for variability modeling where the number of times a feature is included in the configuration can be specified. This operation is described in the literature as feature *cloning* and is handled by *cardinality-based* FMs [30, 13, 25]. Cloning is the ability for a feature and its subtree to be instantiated multiple times and configured differently. However, existing cardinality-based feature modeling approaches support the definition of cardinality for features but do not handle the configuration of such FMs when cardinalities are involved in constraints. In this paper, we address this issue as cloud environment configuration, and therefore SALOON, requires this support.

Indeed, the presence of several instances of the same feature may have an impact on the number of instances for another feature. For example, instances of a given feature may be added, removed or required at a given amount according to the presence or not of instances for another feature. To face this important limitation, we propose new expressions that support the definition of cardinality in constraints. In this paper, we extend a previous investigation in the domain of cardinality-based constraints that was a first step in this way [28]. We clearly state the required expressions and propose an abstract model to define cardinality-based constraints in cardinality-based FMs. With such expressions, the number of feature instances to configure can be precisely defined. We also provide a tool support, implemented in the SALOON framework, to automatically check the validity of these configurations.

Throughout the paper, we illustrate the practical usage of our approach through case studies dealing with the configuration of cloud computing environments. To evaluate the correctness of our approach, we modeled and analyzed a corpus of 10 cloud environments, to *(i)* see how often our proposed expressions occur and validate the need for a cardinality-based constraints support and *(ii)* measure the overhead resulting from the addition of our new cardinality-based constraints in the FM configuration.

The remainder of this paper is organized as follows. In SEC. 2, we discuss the motivation behind the presented approach. SEC. 3 presents the proposed metamodel to define cardinality-based constraints, together with the related tool support. We describe in SEC. 4 the evaluation we lead to empirically assess our approach and we discuss different concerns regarding its validity. In SEC. 5, we describe close-related work in the literature and highlight the current lack of support for cardinality-based constraints in existing feature modeling approaches. Finally, SEC. 6 concludes the paper and presents the perspectives of our work.

2 Motivation

In the cloud computing paradigm, computing resources are delivered as services. Such a model is usually described as *Anything as a Service* (XaaS or *aaS), where *anything* is divided into layers from *Infrastructure* to *Software* including *Platform*. This layered model offers many configuration and dimension choices, for the application to be deployed as well as the configurable runtime environments [24]. At the IaaS level, the entire software stack running inside the virtual machine must be configured as well as the infrastructure concerns: number of virtual machines, amount of resources, number of nodes, SSH access, database configuration, etc. Regarding platforms provided by PaaS clouds, *e.g.*, OpenShift [2], the configuration part only focuses on software that compose this platform: which database(s), application server(s), compilation tool, libraries, etc. The software stack management process is entirely handled by the PaaS provider. Thus, deploying an application on a PaaS has become very trendy [19, 11].

In previous work, we introduced SALOON, a configuration framework together with a methodology to select the adequate cloud environment [27]. SALOON aims at helping stakeholders involved in cloud configuration or deployment by providing a way to deal with the wide range of different services provided by the plethora of available cloud offerings. Thus, SALOON is used as a decision-making tool to determine whether or not a cloud environment is able to host a given application, regarding its provided services together with the application requirements. To handle cloud variability, it relies on FMs, which are used to describe cloud environments and are stored in the SALOON FMs repository. More precisely, SALOON relies on cardinality-based FMs, where each FM describes one given cloud environment. However, supporting cardinality-based constraints related to these FMs has not been addressed yet, which is required, among others, for cloud configurations. We thus provide this support in SALOON, whose more details about this framework can be found in [27].

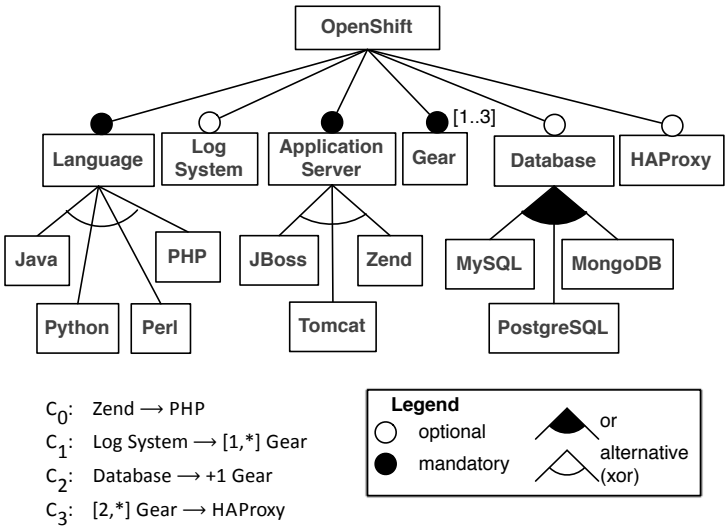


Figure 1: OpenShift cloud feature model (excerpt)

FIG. 1 depicts the OpenShift cloud environment FM we will use as a motivating example throughout the paper to illustrate the different concerns of our approach. The OpenShift

PaaS service configuration[1] includes selecting among several `Application Servers`, `Languages` and `Databases`, if required. These services are deployed on `Gears`. Each `Gear` is given a fair allocation of CPU, memory, disk, and network bandwidth and OpenShift provides up to three `Gears`. `Application Servers` and `Databases` get their own `Gear`, while the logging service relies on one existing `Gear`. More precisely, it means that adding a `Database` in the configuration requires adding one dedicated `Gear` (constraint C_2), while configuring the `Log System` only requires that at least one `Gear` is already configured and is shared (constraint C_1). Horizontal scaling for applications is accomplished using `HAProxy` as a load balancer that routes traffic between `Gears`. The load balancer is automatically configured if there is at least two `Gears` running (constraint C_3). This highlights the fact that, in addition to well-known constraints like C_0, there is a need for additional constraints handling feature cardinality by providing new capabilities such as operators and ranges, *e.g.*, constraints C_2 and C_3.

Therefore, the aim of this paper is to *(i)* provide a support for new means of expression regarding cardinality in constraints, *(ii)* show how often cardinality-based constraints occur in cloud environments configuration, where these new capabilities are required, and *(iii)* provide a tool support to automate the configuration of such FMs. In previous work, we made a first attempt to define a support for cardinality-based constraints in FMs [28]. However, there is a lack in the proposed syntax and semantics to support such constraints, and the tool description is incomplete. To overcome this, we extend this previous work as described in the following section.

3 Cardinality-Based Expressions

Our recent work in the IP PaaSage project [3] regarding deployment of cloud applications and configuration of cloud environments provides evidence that support for expressing constraints in terms of cardinality and reasoning about this cardinality has become necessary. To face this issue, we provide with SALOON such a support as an abstract model, named SALOON_{MM}, depicted by FIG. 2, where $\text{SALOON}_{MM} = \text{INIT}_{MM} + \text{CARD}_{MM}$, as described below. Each cloud FM stored in the SALOON repository conforms to this metamodel. Metaclasses drawn in dotted line in SALOON_{MM} are well-known in the variability modeling community, but may have different names. We refer to this part of the SALOON_{MM} as INIT_{MM}. This INIT_{MM} metamodel remains valid for most feature modeling languages and tools that handle boolean FMs as well as cardinality-based FMs. More precisely for the latter, INIT_{MM} is used in the literature [8, 13, 14] to define a graphical notation for features with cardinality, but there is actually no tool able to handle cardinalities properly during the reasoning and verification stages, in particular regarding constraints.

Therefore, we provide CARD_{MM}, an extension of INIT_{MM} to support both modeling and configuring cardinality-based FMs, depicted as solid line metaclasses in FIG. 2. This extension can be plugged in any existing FM metamodel, *e.g.*, [8, 14], and relies on the `Requires` constraint which allows variability modeler to define cardinalities for both features and constraints. Thus, the number of configured feature instances in the product is not only counted regarding feature cardinalities, but also takes into consideration the fact that a given number of feature instances may require a certain amount of instances for another feature. We describe in the following sections the new expressions we propose to support this kind of constraints, and we illustrate them with constraints from FIG. 1.

[1] The environment described in this paper is the OpenShift *Online* public cloud.

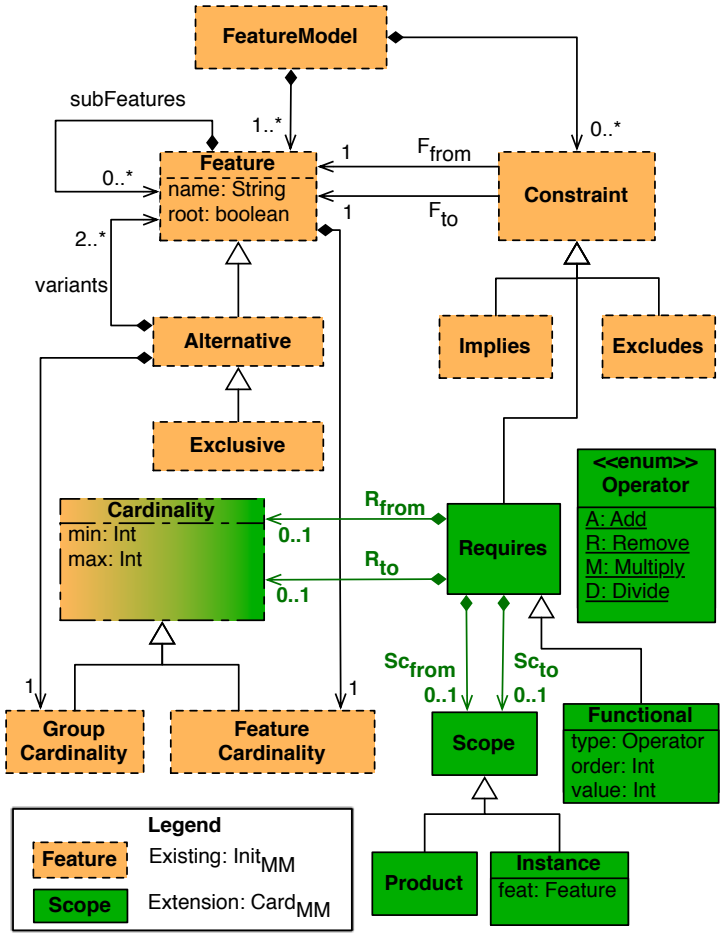


Figure 2: SALOON$_{MM}$

3.1 Ranges in Constraints

Using the `Requires` constraint, one can specify ranges to express constraints over the set of instances for a feature, defined as *range-based* constraints in the following. The constraint C_3 illustrates the use of a range. It describes the fact that if there is at least two instances of `Gear` configured, `HAProxy` must also be configured, to be used as load balancer. These ranges are defined using the `Cardinality` metaclass, through the R_{from} and R_{to} relationships and are related to the F_{from} and F_{to} features respectively. The bound values given in these ranges using the *min* and *max* attributes can be different from the `FeatureCardinality` a feature may have. For example, in the constraint C_3, the range R_{from} `[2,*]` is different from the cardinality `[1,3]` the `Gear` feature holds. Naturally, both R_{from} and R_{to} ranges can be defined within the same constraint. R_{from} and R_{to} relationships being optional (0 as minimum value for their multiplicities), none, one or both of these ranges can be defined within the same constraint.

Let us now also consider constraint C_1, which means that the `Log System` requires at least one `Gear` to run properly. Like in C_3, there is no upper bound defined for the C_1 range R_{to}, but an infinite value represented as *. Although infinite bounds are not permitted in the definition of feature cardinalities, it is possible to use it in the `Requires` constraints description. Indeed, current reasoning and verification tools rely on solvers that do not support infinite value for maximum cardinality. However, defining constraints expressing the presence of *at least n* elements whatever the upper bound remains possible within these solvers, and SALOON relies on such a support to handle range-based constraints.

Finally, R_{from} and R_{to} can be expressed as a single number specified using the *min* and *max* attributes with the same value. For instance, the constraint $[2, 2]$ $A \rightarrow [3, 3]$ B means that if exactly two instances of A are configured, then exactly three instances of B must also be part of the final configuration. Constraint C_2 depicts the use of a value too, but this constraint introduces a new type of expression, based on the use of an *operator*, where the amount of required instances must be defined, as explained below.

3.2 Operators on Feature Instances

Constraints involving an operator (+, -, * or /) are used when the F_{from} feature requires a given amount of F_{to} instances to be added in the configuration. We define this type of constraints as `Functional` constraints, described as *operator-based* constraints in the following. Constraint C_2 depicts the use of an operator, since the `Database` requires one `Gear` to be added. For the sake of clarity, square brackets are not used in such a case. Moreover, C_2 is a particular case where the number 1 appears, otherwise it is not necessary to explicitly define this value, *e.g.*, with C_0. Constraint C_2 is thus a functional one, meaning that each `Database` instance configured requires its own dedicated `Gear`. If a second `Database` is configured, another `Gear` instance must be added and configured to host this database, and both `Gear` instances are not shared between `Database` instances. Thus, each time a `Database` instance is configured, one more `Gear` is required in the configuration. In existing approaches, this kind of constraint can not be properly handled. Suppose for instance that the OpenShift FM contains the two followings constraints, `Database` $\rightarrow$ `Gear` and `ApplicationServer` $\rightarrow$ `Gear`. Then, there will be only one `Gear` present in the final product, and there is no way to configure two of them, whatever the number of `Database` instances is.

As a notation to express that the constraint is applied *for each* feature instance, we introduce an apostrophe in the constraint, near the affected feature name, *e.g.*,

$$A' \rightarrow +n\ B$$

meaning that for each instance of `A`, n more instances of `B` must be added in the configuration.

Several functional constraints may be involved in the FM configuration and have to be properly handled by the underlying tool, to configure the right amount of feature instances. Indeed, there is a particular situation where functional constraints may be given an order of composition. In such a case, two constraints handled in different orders may yield different results. Those constraints thus need to be processed according to a given order of priority. This situation occurs when *(i)* two (ore more) functional constraints, let us take as example Fu_i and Fu_j, implies the same F_{to} feature and *(ii)* $\mathrm{Op}_i \neq \mathrm{Op}_j$ and $\mathrm{Op}_i \in \{+, -\}, \mathrm{Op}_j \in \{*, /\}$, with Op_i and Op_j the operators of Fu_i and Fu_j respectively.

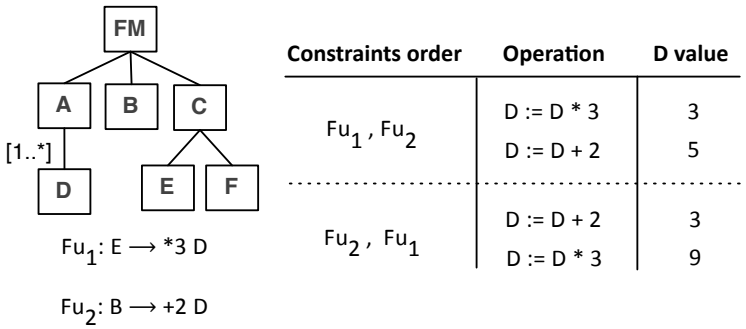


Figure 3: Considering order in Functional constraints

FIG. 3 depicts this situation. In such a case, the number of configured `D` instances differs according to the specified order, whether Fu_1 is considered before Fu_2 or not. To handle this situation, an order of priority can be specified via the *order* attribute defined in the `Functional` metaclass. In SALOON, ordered functional constraints are then computed to yield an unique constraint defining the number of F_{to} instances required. For example, if Fu_1 and Fu_2 are computed with order 1 and 2 respectively, the yielded constraint is

$$\mathtt{B} \wedge \mathtt{E} \rightarrow [9, 9]\ \mathtt{D}.$$

Such a constraint is then well-handled by existing solvers, as described in TABLE 1, SEC. 3.5.

3.3 Scope

When checking the validity of a cardinality-based FM configuration, reasoning and verification tools need to know how to handle the ranges defined in the constraints. Indeed, regarding the way FMs are modeled, `Requires` constraints may involve the number of features instances in different ways, as depicted by FIG. 4.

In the first case a), there is no ambiguity. If there are two instances of `B` configured in the product, then `C` must also be configured in the final product. When looking at case b), the meaning of the constraint C_4 turns out to be unclear. Does C_4 have to be satisfied for configuration (1) or (2)? The answer depends on the way instances are counted. If configured instances are counted regarding the whole product configuration, then C_4 must hold for configuration (1), meaning that since there are two configured instances of `B`, whatever the `A` instance they belong to, then there must be a configured instance of `C`. Conversely, if they are counted regarding a defined feature, then C_4 must hold for each feature instance, as depicted in configuration (2). In such a case, the configured instance of `A` has two configured instances of `B` and must have a configured instance of `C`.

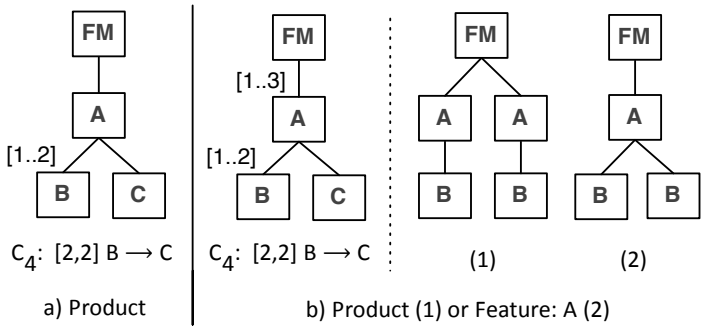


Figure 4: Considering scope in constraints

Our approach resolves the ambiguity by defining the `Scope` a `Requires` constraint is evaluated in. More precisely, a `Scope` can be defined for both R_{from} and R_{to} ranges. For the constraint C_4, the scope is defined (if required) on R_{from}. If C_4 needs to hold for each configured instance of the `A` feature, then the scope is set to `A`, using the *feat* attribute of the `Instance` metaclass. Let us now consider a constraint

$$C_5 : C \rightarrow [2,2]\ B.$$

The scope is here defined for the range R_{to}, to define wether there should be two configured instances of `B` for the whole product or for each instance of `A`, if feature `A` is selected as scope, as for FIG. 4 b) (2). Using the scope, cardinality-based expressions can be defined on a specific range of feature instances, or on the whole one. For example, let us now consider that two constraints

$$C_{sc1} : [1,*]\ B \rightarrow C$$
$$C_{sc2} : [2,2]\ B \rightarrow D$$

are defined for the FM depicted in FIG. 4 b), with feature `A` defined as scope. Then, all `A` instances are affected by constraint C_{sc1} while only a given range of them (those with exactly 2 configured instances of `B`) are affected by C_{sc2}.

The notion of `Scope` we consider here differs from the one described by Michel *et al.* [25], that was dealing with the semantics of feature cardinalities according to the way FMs are modeled. The scope was either *Clone* or *Feature*, and used in the presence of feature cardinality to define if the instances of features should be counted rather than the features that were instantiated.

3.4 Semantics

The above-described cardinality-based expressions are specified through the `Requires` and `Functional` metaclasses and offer a support to precisely define how constraints are expressed in terms of feature instances to be configured. Considering that:

- $\mathcal{M} = (\mathcal{F}, \varphi)$ is a cardinality-based FM, with $\mathcal{F}$ its non empty set of features and φ its set of constraints,
- $\omega : \mathcal{F} \rightarrow \mathbb{N} \times \mathbb{N}$ indicates the cardinality of each feature $(\forall f \in \mathcal{F},\ \omega(f) = [n, m])$,
- $\lambda : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is an operation, where $\lambda(a, b) = a\ \delta\ b$ with $\delta \in \{+, -, *, /\}$ and $a, b \in \mathbb{N}$,

- $\forall f \in \mathcal{F}$, `card`$(f)$ returns the number of instances of f,

we formally define a `Requires` constraint ρ and a `Functional` constraint ρ^+ as follows (with ρ, $\rho^+ \in \varphi$).

Definition 1. [CONSTRAINTS]
Let $\rho = (C_{from}, F_{from}, C_{to}, F_{to})$ be a `Requires` constraint and $\rho^+ = (C_{from}, F_{from}, \delta, N_{to}, F_{to})$ a `Functional` one, where

- $F_{from}, F_{to} \in \mathcal{F}$ where $F_{from} \neq F_{to}$, $N_{to} \in \mathbb{N}$;
- C_{from}, C_{to} are cardinalities. Each one defines an interval [i-j] with i, j $\in \mathbb{N}$ and i $\leq$ j or a value k_{from} (respectively k_{to}), with k_{from}, $k_{to} \in \mathbb{N}$. C_{from} (respectively C_{to}) is the range over the set of clones for F_{from} (respectively F_{to});

Then ρ is satisfied if

$$\vee \begin{cases} \omega(F_{from}) \subseteq C_{from} \\ \mathit{card}(F_{from}) = k_{from} \end{cases} \Rightarrow \vee \begin{cases} \omega(F_{to}) \subseteq C_{to} \\ \mathit{card}(F_{to}) = k_{to} \end{cases}$$

and ρ^+ is satisfied if

$$\vee \begin{cases} \omega(F_{from}) \subseteq C_{from} \\ \mathit{card}(F_{from}) = k_{from} \end{cases} \Rightarrow \lambda(\mathit{card}(F_{to}), N_{to})$$

ρ and ρ^+ hold in a *product* scope. In the case of a feature f is given as scope, then $\forall\{f_i...f_n\}$ instances of f, ρ and ρ^+ must hold.

Adding such cardinality-based expressions to the definition of cardinality-based FMs changes the way their configurations are checked. In addition to the well-known declarative constraints where the selection of a feature implies or excludes the selection of another one, *e.g.*, C_0, these constraints must hold for a configuration to be valid.

3.5 Tool Support

To handle such expressions and properly reason about the configurations of cardinality-based FMs, the approach proposed in this paper is implemented into the Java-based SALOON framework [27]. Each cloud environment is described as a cardinality-based FM which conforms to our proposed SALOON$_{MM}$ metamodel and is stored in a FMs repository. To define SALOON$_{MM}$, our approach relies on the *Eclipse Modeling Framework* (EMF) [32], which is one of the most widely accepted metamodeling technologies. The EMF provides, among others, code generation facilities to *(i)* produce a set of Java classes for the metamodel used in the Java API of the SALOON framework and *(ii)* generate graphical editors used to create dynamic instances of SALOON$_{MM}$, *i.e.*, cardinality-based FMs and theirs constraints. SALOON$_{MM}$ is thus described as an *ecore* file while dynamic instances are defined as XMI models. The XMI format is used to support model persistence in the FMs repository.

SALOON loads each XMI model and parses it to generate the corresponding set of constraints, thus representing the FM as a *Constraint Satisfaction Problem* (CSP) model. CSP solvers are well-suited to reason about cardinality-based FMs, and the translation of these FMs to CSP is well-known [23, 8, 4]. Thus, each feature X from the FM is translated into an integer variable

X' describing the number of instances to be configured in the final product. We extend this translation to handle cardinalities-based FMs by supporting the cardinality-based expressions we propose in this paper. Rules for translating `Requires` and `Functional` constraints to constraints handled in CSP are listed in TABLE 1.

FM notation	**CSP constraint**
[i,j] A $\rightarrow$ [n,m] B	ifThen(A' in {i,j} ; B' in {n,m})
A $\rightarrow \delta$n B	ifThen(A' $>$ 0 ; B = B δ n)
A $\wedge$ B $\rightarrow$ [n,*] C	ifThen(and(A' $>$ 0 ; B' $>$ 0) ; C $\geq$ n)

Table 1: Transformation rules from FM constraints to CSP constraints

Thus, a feature X that has *at least* i configured instances and *at most* j ones, is translated to an integer variable X' whose value has to be in the same $\{i,j\}$ range for the constraint to hold. When the maximum number of instances for a feature X is not defined (noted as *), then the constraint holds if the value of X' is greater than the X lower bound, whatever its value is (assuming that X is not involved in other constraint).

Once FMs are translated, SALOON relies on the off-the-shelf Choco CSP solver [21] to reason on these FMs, *e.g.*, calculating the number of valid configurations, checking if a configuration is valid regarding a given set of select features or detecting dead features. We thus provide with SALOON a mean to handle entirely cardinality-based FMs, from feature modeling to model configuration and verification.

4 Evaluation

In this section, we describe the experiments we conducted to evaluate our approach. This evaluation aims at investigating the following research questions:

R1: *Soundness.* Is SALOON, and the cardinality-based expressions in particular, well-suited to support cardinality-based feature modeling?

R2: *Scalability.* Is our approach applicable for cardinality-based FMs with many features and operator or range-based constraints?

First, we empirically evaluate the soundness of our approach by using the SALOON_{MM} as support to define a substantial number of cloud environments. Secondly, we analyze the overhead that results from *(i)* the use of our cardinality-based expressions and *(ii)* the translation from XMI models to CSP constraints. Finally, we discuss the threats to validity of our approach.

4.1 Methodology

Our evaluation is based on the study of 38 clouds environments, whose list can be found in [1]. Among those 38 clouds, we selected 10 of them, each one then being modeled as a FM which conforms to SALOON_{MM}. We define this set of 10 cloud FMs in the following as the $Cloud_{corpus}$. This selection is based on the following criteria:

- *Representativeness.* Both IaaS and PaaS clouds environments are represented in the $Cloud_{corpus}$. Thus, we cover a broader range of cloud providers and show that our approach

is well-suited whatever the cloud layer involved is. Moreover, we select both well-known and less-known cloud providers, *e.g.*, Windows Azure and Jelastic respectively.

- *Data access.* We select clouds whose features are easily accessible either through a web configurator or in the technical documentation. Indeed, a major issue when modeling cloud environments is to find their provided functionalities drawn in the huge amount of information they spread.
- *Variability factor.* The variability factor (VF) is a relationship between the number of valid configurations and 2^n where n represents the total number of leaves in the FM. If the VF is too close to either 0 or 1, it means that the feature model is too restricted or too flexible respectively. We selected clouds whose related FM's VF is as close as possible to 50%.

TABLE 2 shows the set of cloud environments we used in our empirical evaluation. For each application, the table shows the cloud environment name (`Cloud`), its type (`Type`), the number of features defined in the related FM (`Feat`), the number of constraints (`Cons`) and the variability factor (`VF`). All the corresponding FMs can be found in [1].

Cloud	**Type**	**Feat**	**Cons**	**VF (%)**
`Cloudbees` (CB)	PaaS	32	7	52
`CloudFoundry` (CF)	PaaS	37	15	61
`Dotcloud` (Do)	PaaS	31	83	34
`GoGrid` (GG)	IaaS	23	21	64
`Google App Engine` (GAE)	PaaS	33	10	57
`Heroku` (He)	PaaS	51	20	38
`Jelastic` (Je)	PaaS	31	6	48
`OpenShift` (Op)	PaaS	33	16	27
`Pagoda Box` (Pa)	IaaS/PaaS	31	8	46
`Windows Azure` (Win)	IaaS/PaaS	59	47	35

Table 2: Modeled cloud environments

We use the variability factor as a criteria that indicates if the modeled cloud environment is close enough from the reality. Indeed, when modeling the cloud environment, hidden features or constraints may have been forgotten. A FM whose VF is too restricted means that there are not enough variabilities. On the contrary, a too flexible model means that there are not enough commonalities. The VF factor given in TABLE 2 is based on the presence or not of a feature in the final product, whatever the number of feature instances. We thus calculate this indicator *without* taking into consideration feature cardinalities when computing the number of valid configurations.

During our modeling study of cloud environments, we extract a *template* model that we use for each FM. Thus, under the root feature, each cloud FM defines the following abstract features (when consistent): *(1) Language,* that defines the language the application to be hosted has been implemented in, *(2) Application Server,* the environment which contains and runs the application, *(3) Database,* SQL-based or not, to store the application data, *(4) Resource,* that describes available amount of resources, *e.g.*, RAM or CPU, and *(5) Service,* all the miscellaneous services provided by the cloud environment, *e.g.*, build nodes.

4.2 Results

Having modeled 10 cloud environments based on the approach presented in previous sections, we describe in this one the results of our empirical evaluation regarding the soundness and scalability concerns of this approach.

4.2.1 Cardinality-Based Expressions Occurrence

The aim of this evaluation is to assess the soundness of our approach and determine how often do cardinalities occur in cloud environments FMs, both for features and constraints. More precisely, regarding constraints with cardinality, this evaluation determines what kind of expression is required in which ratio. Thus, TABLE 3 shows the amount of features (`Feat`) and constraints (`Constraints`) with cardinality. For each constraint, it highlights the constraints that require a range (`Ran`) or an operator (`Ope`). The number of features and constraints with cardinalities varies from a cloud environment to another according to the provided services and the way we modeled it (regarding the *data access* criteria described in SEC. 4.1). For instance, `Dotcloud` and `OpenShift` with respectively 26 and 4 features with cardinality, are two cloud platforms very similar regarding services they provide. However, the way they can be configured differs: the `Dotcloud` configuration is more resource-oriented while the `OpenShift` one is more dedicated to provided functionalities or support.

		Constraints		
Cloud	**Feat**	**Total**	**Ran**	**Ope**
`Cloudbees`	3	7	7	6
`CloudFoundry`	5	7	1	6
`Dotcloud`	26	83	83	83
`GoGrid`	8	21	21	21
`Google AE`	7	10	10	10
`Heroku`	5	14	0	14
`Jelastic`	3	5	1	4
`OpenShift`	4	16	6	6
`Pagoda Box`	8	8	4	8
`Windows Azure`	11	47	47	47

Table 3: Cardinality occurrence in the $Cloud_{corpus}$

On the whole, regarding the $Cloud_{corpus}$, there are 80 features with cardinality and 218 constraints based on our cardinality-based expressions, which gives an average FM with 8 features with cardinality and about 22 cardinality-based constraints. 90% (9/10) of clouds require range-based constraints while 100% of them require operator-based constraints. Some FMs rely on ranges and operators in 100% of cardinality-based constraints, while `Heroku` and `Openshift` are those with less range and operator-based constraints (0% and 37% respectively.) In average, 82% (180/218) of cardinality-based constraints are range-based ones and 94% (205/218) are operator-based ones.

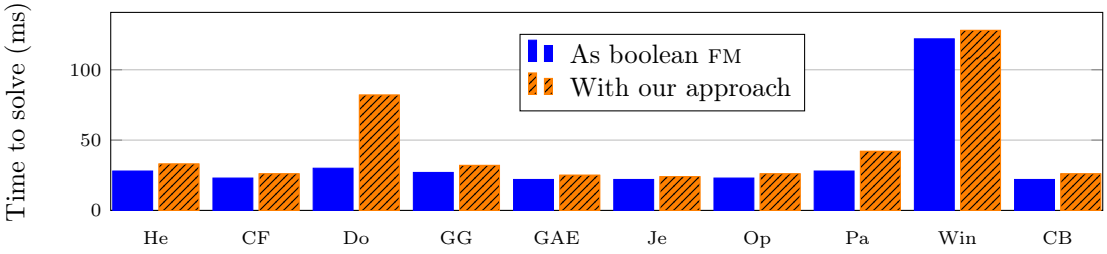


Figure 5: Verification time overhead

Observation. We note an interesting and unexpected result raising from this evaluation. When looking into the details of the constraints using an operator-based expression [1], we can see that these constraints always occur when dealing with resources allocation, such as the constraint C_2. These constraints mean that the F_{from} feature (or its instances) involved in the constraint requires the corresponding amount of resources (*i.e.*, of F_{to} instances) to run properly. There is actually two ways to interpret these constraints, related to the idea of *shared* resources. Let us now consider the constraint

$$C_6 : Feat \rightarrow +2\ Res,$$

where *Res* is a resource feature, *e.g.*, a CPU block. The first and intuitive way to understand C_6 is that adding two CPU blocks in the final product makes *Feat* run properly. However, in most of cases we found, the underlying meaning of this kind of constraints was that *Feat requires 2 dedicated CPU block to run properly.* In those cases, the targeted resources are not shared among the product features. At feature modeling and configuration stages, this semantic difference does not have any importance since in both cases, feature *Res* must have two more instances configured. Nevertheless, to properly handle such operator-based expressions in the whole SPL engineering process, a clear semantics of these constraints is required in the *derivation* process, where concrete artifacts are built together to yield the final software product. This key point is one possible direction for future work.

To summarize, while we can not yet conclude that our approach can be generalized to every domain with variability, results raising from this evaluation show that it remains well-suited for cloud environment modeling, while state-of-the-art approaches do not provide such a support.

4.2.2 Scalability

The aim of this evaluation is to show that SALOON can be used as any existing feature modeling framework, *e.g.*, FaMa[2], S.P.L.O.T.[3] or FeatureIDE[4]. As these frameworks do not support the cardinality-based expressions we propose in this paper and rely on different solvers, we cannot directly compare the verification time between them and SALOON. We thus carried out experiments to point out the practicability of our approach.

This evaluation is twofold. First, we measure the overhead that results from the addition of the `Requires` and `Functional` constraints in the verification time of the underlying CSP solver. This evaluation aims at showing that the time to solve the models does not grow significantly with FMs modeled with the extension we provide. Second, we carried out further experiments

[2] `http://www.isa.us.es/fama/`
[3] `http://www.splot-research.org/`
[4] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`

to measure the translation time from XMI format (when FMs are instance of SALOON$_{MM}$) to constraints handled by the CSP solver. This translation process, neither part of the feature modeling nor the configuration one, may be a threat to practicability of SALOON if taking too much computation time.

We performed our evaluation on a MacBook Pro with a 2,6 GHz Intel Core i7 processor and 8 Go of DDR3 RAM. For the first evaluation regarding the overhead that may be caused by the additional verifications due to our cardinality-based expressions, we modeled each cloud environment twice. In the first version, they are modeled using our approach, with `Requires` and `Functional` constraints, as depicted in [1]. In the second one, they are modeled as boolean FMs, without any cardinality neither for features nor for constraints. Thus, only `Implies` and `Excludes` constraints are used. Therefore, the semantics of the FM regarding the modeled cloud environment is inappropriate but is not taken into consideration for this evaluation. We then compare the time taken by the solver to find if there is a valid configuration for both versions.

Each model verification run was repeated 100 times to avoid random variations and we use in this evaluation the average time we computed for each model. As illustrated by FIG. 5, the support for our cardinality-based expressions generates a small increase in the required time to find a solution. Highest and lowest gaps are obtained for the `DotCloud` and `Jelastic` models, with 52 (+63%) and 2 (+8%) milliseconds respectively. The difference regarding the `DotCloud` environment is due to the high number of constraints with ranges and operators it holds (83). Since these constraints are transformed to `Implies` constraints in the boolean FM, the solver takes much less time to reason about the configurations. Moreover, there are many features with cardinality (26), which leads to many range value checking in FMs relying on the cardinality-based expressions. Although we did not define a threshold for this experiment, we can fairly argue that the overhead that results from using our cardinality-based expressions is not a major threat to practicability of our approach and, by extension, of SALOON, since it is almost imperceptible.

For the second evaluation regarding the translation time from XMI to CSP constraints, we developed an algorithm that, given the number of features and constraints, generates a random cardinality-based FM. Namely, it generates features with cardinalities and constraints relying on the expressions described in SEC. 3. We generate random FMs with 10, 50, 100, 500, 1000, 5000 and 10000 features. For each FM, the number of constraints to be generated changes from 10%, 20%, 30% to 40% of the number of features, to be as close as possible of a real FM. The type of each constraint (`Implies`, `Excludes`, `Requires` or `Functional`) is randomly assigned on the fly. Each run is thus repeated 4 times (one for each ratio of constraints), and 1000 FMs are randomly generated by run, thus leading to a total of 1000 * 4 * 7 = 28000 generated FMs. We then compute the average time of the 4000 generated FMs for the given amount of features.

Nb Feat.	10	50	100	500	1000	5000	10000
Time (ms)	1,7	1,9	2,3	5,4	9,6	207	783

Table 4: Time to translate from XMI to CSP

As shown by TABLE 4, the translation time from XMI to CSP constraint is from 1,7 to 783 ms for 10 to 10000 features respectively. This time is slightly increasing with the size of the model, in particular with FMs composed of less than 1000 features. For FMs with more features, it then increases faster and tends to keep increasing this way. However, we believe that it is not a major threat to scalability for the two following reasons. First, the bigger FM from the $Cloud_{corpus}$ contains "only" 59 features (`Windows Azure`). Moreover, most of existing FMs contain less than

500 features, *e.g.*, those from the S.P.L.O.T. repository. Then, one of the biggest existing FM, which is the Linux feature model, has over 5000 features [31]. This translation time overhead remains therefore fairly low and does not hinder the usability of the SALOON framework.

Overall, as our empirical evaluation shows, we observe that SALOON is well-suited to handle the cardinality-based expressions we propose in this paper, both for feature modeling and automated configuration verification. Indeed, the results show that *(i)* most of constraints we met required cardinality-based expressions (range-based 62%, operator-based 86%) and *(ii)* SALOON generates a negligible overhead when checking the related configurations.

4.3 Threats to Validity

There are several concerns in our evaluation that may form threats to validity. We consider these concerns from two perspectives.

External validity. The choice of the $Cloud_{corpus}$ threatens external variability. We first selected a substantial number of cloud providers from many sources (our knowledge, case studies in cloud related conference papers, web comparator, etc.). But the intrinsic nature of cloud computing, in particular the variability of cloud environments, was a limiting factor. Cloud environments provide a shared pool of highly configurable computing resources, in several configuration levels. It was thus impossible for us to fully reify the cloud environments and we had to limit our feature modeling to features which are explicitly released by cloud providers, since implicit features and constraints finding and modeling are far more complex. FMs used in our experiments were thus not exhaustive. Moreover, the number of cloud environments studied in the $Cloud_{corpus}$ is relatively low regarding the real number of existing providers. However, as explained in SEC. 4.1, we try to be as fair as possible when selecting our case studies for them to be representative of the domain. Finally, due to the evolutive nature of cloud computing, *e.g.*, cloud providers that appear/disappear or existing environments evolving, $Cloud_{corpus}$ case studies might not be valid over the long term.

Internal validity. There are also threats regarding internal variability. The metamodel we propose has been defined based on the findings we did in our previous work regarding cloud environment configurations [27]. We can thus assess the validity of our approach for this particular domain and, while we cannot claim that our results represent cardinality-based FMs modeling and configuration in a broader way, we believe that it can be used to configure cardinality-based FMs in others engineering domain. Other constraint expressions may be required, like constraints involving one specific feature instance, which we do not provide (even if using the `Scope` support is actually a possible way around). However, as our approach is an extension of existing feature modeling ones, it can itself be extended to provide those new expressions and handle cardinality-based feature modeling and configuration in other domains. Another threats to internal validity arise from the `Functional` constraints based on the *multiply* operator. Let us take as example the following constraint C_x: `R` $\rightarrow$ *n `S`. If `S` is an optional feature, then its value when handled by the CSP solver is 0. Then, whatever the value of n is, the number of configured instances for feature `S` will always be 0. This constraint thus properly works only with mandatory features. Nevertheless, within each cloud FM of the $Cloud_{corpus}$, we never met this particular case and we do not think it could possibly happen.

5 Related Work

Several works were proposed to deal with the problem of cardinality-based FM modeling and configuration. We describe in this section the most significant close-related approaches in this area.

Cardinality-based FMs were first introduced by Riebisch *et al.* [30] where UML *multiplicities* were used as an extension to the original FODA notation to allow features to be instantiated several times. The authors introduced the notion of feature cardinality and proposed a syntax to define this cardinality in the FMs. Several authors [8, 13] then proposed a metamodel for cardinality-based FMs. These metamodels can easily be extended with the approach proposed in this paper to handle the configuration of cardinality-based FMs. Czarnecki *et al.* also defined a new semantics for this kind of FMs, in particular regarding the group feature cardinality [13]. Michel *et al.* then defined their own semantic domain, but none of them described a semantics for constraints over these cardinality-based features [25]. Czarnecki *et al.* went further in their research by proposing to use languages like OCL to define constraint for FDs with cardinalities [15]. Although OCL is well-suited to define constraints in a general way, it is not really dedicated to FMs and it does not handle the expressions proposed in Sec. 3 regarding features and constraints with cardinality. Trinidad *et al.* proposed the FAMA tool suite for automated analyses of feature models [33]. Although features with cardinality can be modeled with FAMA, this framework does not handle the configuration of such FMs when cardinalities are involved in constraints.

Zhang *et al.* presented a BDD based approach to verify constraints with cardinalities, based on their own semantics of cloning [36]. They described their two different constraint patterns and the way they can be verified but did not provide any abstract syntax to define such FMs. In Gomez *et al.* [20], the authors present their own metamodel and the way they rely on model-driven engineering to configure their FMs. They introduce a new kind of constraint denoted as *Use* where a feature A can use a given amount of feature B instances. The approach we propose in this paper goes in the same direction, but we go further in cardinality-based constraints support, in particular regarding ranges and operators. Dhungana *et al.* propose an approach based on generative constraint satisfaction problems to handle constraints in the context of cardinality-based FMs [16]. They also introduce a notation to express ranges in constraints over the set of feature instances, as we do in this paper, but do not handle other constraint expressions and do not provide any tool support. More recently, Cordy *et al.* proposed their own language *TVL** to handle features with attributes and multi-features, *i.e.*, features that can be instantiated several times [12]. They gave its formal semantics and defined two new constraints to handle the configurations. However, *TVL** does not provide any support to specify ranges and operators for constraints with cardinality.

Several studies rely on feature models in the domain of cloud computing. FMs are used to capture virtual machine configurations and energy consumption optimization [29, 17] or as a support in a cloud service selection process [34]. The approach presented in this paper addresses limitations faced in these works regarding cloud services feature modeling and configuration checking. FMs have also been described as being exactly on the right level of granularity and abstraction for representing the particularities of a cloud service [35]. Finally, Frey *et al.* proposed an approach based, among others, on constraint satisfaction (*e.g.*, "*if a grow rule exists in the configuration, a shrink rule also has to be present*") to find the most adequate IaaS-based cloud environments [18]. Our approach is not specific to this cloud layer and relies on FMs, a dedicated support to reason on configurations and their constraints.

6 Conclusion

Current feature modeling approaches do not support configuration in the presence of feature cardinalities. More precisely, several feature instances can be specified but no constraints can be expressed on these cardinalities, even though the presence of several instances for a given feature may have consequence on the presence or not of another feature (or its instances). In this paper, we face this limitation by providing new cardinality-based expressions, in particular ranges and operators over the set of instances for a feature as well as the definition of a scope for these constraints to be properly handled. Our approach is based on an abstract model, SALOON$_{MM}$, and we define its formal semantics. These cardinality-based expressions are implemented in the SALOON framework, which relies on an off-the-shelf CSP solver to automate the verification of configurations from FMs that conform to our abstract model.

To evaluate our approach, we investigate feature modeling in the domain of cloud computing environments, where constraints based on cardinality are required to properly support their configurations. By analyzing 10 different clouds, we provide empirical evidence on the soundness and scalability of our approach, and we believe that it can be used to configure cardinality-based FMs in other engineering domains. Moreover, as the proposed abstract model is an extension of the existing ones regarding feature modeling, it can be used as support in further studies to handle new constraint expressions required in the modeling and configuration of cardinality-based FMs.

Besides, we plan for future work to further study the configuration of FMs with the presence of cardinality in constraints. These experiments should include the analysis of software systems from different domains to identify new constraint expressions to be supported by our approach. Another possible direction is the management of such configurations during the SPL derivation process, which, to the best of our knowledge, has not been addressed yet.

References

[1] `http://researchers.lille.inria.fr/~cquinton/research/rr-8478.html`.

[2] The Open Hybrid Cloud Application Platform by Red Hat. `https://www.openshift.com`.

[3] PaaSage: Model-based Cloud Platform Upperware. `http://www.paasage.eu`, 2013.

[4] C. E. Alvarez Divo. Automated Reasoning on Feature Models via Constraint Programming. Master's thesis, Uppsala University, Department of Information Technology, 2011.

[5] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development, July/August 2009. Refereed Column.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, EECS Department, University of California, Berkeley, 2009.

[7] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Inf. Syst.*, 35(6):615–636, Sept. 2010.

[8] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using Java CSP Solvers in the Automated Analyses of Feature Models. In *Proceedings of the 2005 international conference on Generative and Transformational Techniques in Software Engineering*, GTTSE'05, pages 399–408, Berlin, Heidelberg, 2006. Springer-Verlag.

[9] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Gener. Comput. Syst.*, 25:599–616, June 2009.

[10] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[11] CloudTimes. No Other Cloud Sector is Growing as Fast as PaaS. `http://cloudtimes.org/2013/08/23/451-research-no-other-cloud-sector-is-growing-as-fast-as-paas/`, 2013. Last accessed 12 September 2013.

[12] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-features. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13, pages 472–481, Piscataway, NJ, USA, 2013. IEEE Press.

[13] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[14] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[15] K. Czarnecki and C. H. P. Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *International Workshop on Software Factories at OOPSLA'05*. ACM, San Diego, California, USA, 2005.

[16] D. Dhungana, A. Falkner, and A. Haselbock. Configuration of Cardinality-Based Feature Models Using Generative Constraint Satisfaction. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 100–103, 30 2011-sept. 2 2011.

[17] B. Dougherty, J. White, and D. C. Schmidt. Model-driven Auto-scaling of Green Cloud Computing Infrastructure. *Future Generation Computer Systems*, 28(2):371–378, 2012.

[18] S. Frey, F. Fittkau, and W. Hasselbring. Search-Based Genetic Optimization for Deployment and Reconfiguration of Software in the Cloud. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13, pages 512–521, Piscataway, NJ, USA, 2013. IEEE Press.

[19] Gartner. Hype Cycle for Cloud Computing, 2012. `http://lcolumbus.files.wordpress.com/2012/08/hype-cycle-for-cloud-computing-20121.jpg`, 2012. Last accessed 12 September 2013.

[20] A. Gómez and I. Ramos. Cardinality-Based Feature Modeling and Model-Driven Engineering: Fitting them Together. In D. Benavides, D. S. Batory, and P. Grünbacher, editors, *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*, volume 37 of *ICB-Research Report*, pages 61–68. Universität Duisburg-Essen, 2010.

[21] N. Jussien, G. Rochart, and X. Lorca. Choco: an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, Paris, France, France, 2008.

[22] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) - Feasibility Study. Technical report, The Software Engineering Institute, 1990.

[23] R. Mazo, C. Salinesi, D. Diaz, and A. Lora-Michiels. Transforming Attribute and Clone-enabled Feature Models into Constraint Programs over Finite Domains. In L. A. Maciaszek and K. Zhang, editors, *ENASE*, pages 188–199. SciTePress, 2011.

[24] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology, 2009.

[25] R. Michel, A. Classen, A. Hubaux, and Q. Boucher. A Formal Semantics for Feature Cardinalities in Feature Diagrams. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS'11, pages 82–89, New York, NY, USA, 2011. ACM.

[26] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[27] C. Quinton, N. Haderer, R. Rouvoy, and L. Duchien. Towards Multi-Cloud Configurations Using Feature Models and Ontologies. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, MultiCloud'13, pages 21–26, New York, NY, USA, 2013. ACM.

[28] C. Quinton, D. Romero, and L. Duchien. Cardinality-based feature models with constraints: a pragmatic approach. In *Proceedings of the 17th International Software Product Line Conference*, SPLC'13, pages 162–166, New York, NY, USA, 2013. ACM.

[29] C. Quinton, R. Rouvoy, and L. Duchien. Leveraging Feature Models to Configure Virtual Appliances. In *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*, CloudCP'12, pages 2:1–2:6, New York, NY, USA, 2012. ACM.

[30] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending Feature Diagrams with UML Multiplicities. In *6th World Conference on Integrated Design & Process Technology (IDPT2002)*, June 2002.

[31] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 461–470, New York, NY, USA, 2011. ACM.

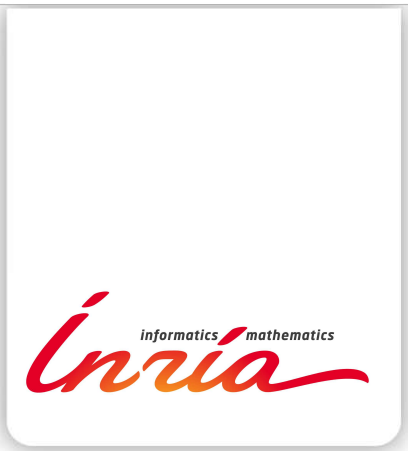
[32] D. Steinberg, et al. *EMF: Eclipse Modeling Framework (2nd Edition).* 2nd revised edition, 2009.

[33] P. Trinidad, D. Benavides, A. Ruiz-Cortes, S. Segura, and A. Jimenez. FAMA Framework. In *12th International Software Product Line Conference, SPLC'08.*, pages 359–359, 2008.

[34] E. Wittern, J. Kuhlenkamp, and M. Menzel. Cloud service selection based on variability modeling. In *Proceedings of the 10th international conference on Service-Oriented Computing*, ICSOC'12, pages 127–141, 2012.

[35] E. Wittern, N. Schuster, J. Kuhlenkamp, and S. Tai. Participatory Service Design through Composed and Coordinated Service Feature Models. In *Proceedings of the 10th international conference on Service-Oriented Computing*, ICSOC'12, pages 158–172, 2012.

[36] W. Zhang, H. Yan, H. Zhao, and Z. Jin. A BDD-Based Approach to Verifying Clone-Enabled Feature Models' Constraints and Customization. In H. Mei, editor, *High Confidence Software Reuse in Large Systems*, volume 5030 of *Lecture Notes in Computer Science*, pages 186–199. Springer Berlin Heidelberg, 2008.



RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399